BIT 8 (1968), 36–42

FEATURES OF THE GIER ALGOL 4 SYSTEM

PETER NAUR

Abstract.

A brief survey is given of the new features included in the Gier Algol 4 compiler system, as compared with its predecessors. As background the more recent additions to the hardware of the machine are described. The new features primarily aim at a more effective use of the existing machine facilities, including optional facilities existing only in some installations.

Introduction.

Since 1962, when the first version of the Gier Algol system was developed, three new versions, incorporating various improvements over the original design, have been completed. Of these the version known as Gier Algol 4, completed in 1967, represents the greatest step forward from its predecessor. The following paper is an informal description of the new features introduced in Gier Algol 4. A full, user-oriented description is given in the manual of the system [1].

Background.

The new features of Gier Algol 4 must be seen on the background of the characteristics of the machine and the previous Gier Algol systems as reported in ref [2]. The main justification for the new version was the extension of some of the Giers with additional units as follows:

1. A so-called buffer store of 4096 words of core store. Because of the limits of the basic addresses of the Gier, this store has some of the characteristics of a peripheral unit, but it does allow access to single words within 100 microseconds.
2. Disk file units of 384000 words or more.
3. Magnetic tape units.

Another justification was the desire to improve the efficiency of the system for general and administrative data processing. In fact, while the system had proved highly useful for scientific, engineering and experimental data processing uses, lack of effective handling of parts of words had prevented its wide adoption for data processing production programming.

Internal improvements.

Certain improvements of the system do not influence the writing of programs directly, but only improve the service to the users offered by the system. In the translator the most important improvement of this kind is the extension of the check of the match of types to cover actual parameters of procedures called directly (but not of those called via actual-formal correspondence).

Internal improvements of the execution phase include that integers are represented internally by 40 bits, including the sign, and not as floating numbers having only 29 bits in the fixed point part. This is valuable both because larger values can be accomodated, as needed in handling monetary amounts, and because of the higher speed of operations in most cases.

A very important improvement is the use of the buffer store for holding subscripted variables. Previously only up to some 700 variables could be used, and execution speed considerations made it desirable to put the limit even lower, at about 500 variables, cf. [3]. With the buffer store 4096 subscripted variables are allowed, without any ill effects on the speed of execution because of the squeezing of the program.

References to subscripted variables during execution were speeded up in two different ways. First the internal representation of array identifiers was made more convenient for use in the subscription process, by supplying the base address of the array elements and the reference to the storage mapping coefficients in two separate computer words, instead of packing them into one word. Thus the translator already distinguishes two locations for each array. This reduces the time of a reference to a variable having one subscript by about 30 percent. The cost is extra complication in case of array identifiers supplied as formal names, which so far could be treated precisely as non-formal array identifiers. This can be solved in several different ways. As one possibility every reference to such a subscripted variable can be done differently during execution, thus using more time. In order to achieve the highest execution speed we have, however, adopted an alternative solution, to let the procedure call during execution make a complete copy of the description of the array into local locations of the procedure body. The number of locations needed for the description depends on the number of subscripts of the array, however. Within the procedure body this number can only be determined from the uses of the array identifier in subscripted variables. Consider, for example, the following program:

```
begin
  procedure Q(A); array A;
  begin array C[3:7];
    ...
    A[5, 7] := C[4] - B[9, 5]
    ...
  end;
  array B[2:66, 4:22];
  Q(B)
end
```

Here the descriptions of the arrays B and C during execution will be generated in locations which are local to the outermost block and the procedure body, respectively, and during translation the references to each part of these array descriptions may be determined exclusively by using the declaration of the arrays. In case of the formal array A this same information can only be obtained during translation by combining the specification and the use of the identifier in a subscripted variable, because only in this latter context is it possible to determine the number of subscripts of the array.

In Gier Algol 4 the translator is charged with the extra task of picking up from subscripted variables the information about the number of subscripts. This required a substantial extension of the translation pass 4, which has the general task of picking up declarative information.

Extensions of Algol 60.

The chief extension of the basic language was the addition of case expressions and case statements, as they have been introduced by C. A. R. Hoare [4, 5].

Minor extensions consisted in adding a number of operators, such as **abs** for taking the absolute value, without change of type, **entier** having the same effect as the procedure entier, **round** for getting the nearest integer value, and **mod** for getting the remainder of the integer division.

In designing Gier Algol 4 it was felt to be imperative to provide means for effective manipulation of parts of words, including parts of real and integer values. Much thought was given to introducing a new type, **general** say, which would allow such manipulation. This possibility was, however, rejected in favor of the following solution. First we extend the concept of values of type Boolean so that each value has 40 bits, the truth value being given by the first of these while the full set may be regarded as a bit pattern. Second, we introduce a way to write literal

Boolean values as bit patterns. For example, **7 21 3 3** will denote a bit pattern consisting of 7 bit positions holding the binary representation of the decimal value 21, followed by a pattern of 3 positions holding the value 3, the remaining positions set to 0, thus:

0010101 011 00000 00000 00000 00000 00000 00000 .

Third, the logical operators will work on all 40 bits of patterns in parallel, with the interpretation: 0 is **false**, 1 is **true**. Fourth, we introduce one new operator, **shift**, the expression

p **shift** *q*

yielding as result a Boolean value obtained by shifting cyclically the bit pattern given as the value of the Boolean variable *p* a number of positions given as the integer value of *q*. Fifth, to take care of the free manipulation of values of other types we allow the symbols **integer**, **real**, **Boolean**, and **string** to be used as operators. Each of them may operate on an operand of any type and will yield as result a value of the type indicated by the operator itself. The action of the operators is defined in most cases by saying that the internal bit representation of the result is the same as that of the operand. Thus these operators usually require no execution time. Their chief effect is to suppress the alarm for mismatch of types which would occur during translation if they were omitted.

As an illustration the following statement extracts the exponent of the real variable *R*, i.e. the first 10 bits, and assigns them to the integer *I*:

$$I := \textbf{integer}(\textbf{10}\ 1023 \wedge \textbf{Boolean}\ R\ \textbf{shift}\ -30)$$

Data in backing store.

The addition of disk file units to the machine makes it practical to store data in a semi-permanent manner, i.e. beyond the execution period of individual programs. The new Algol compiler therefore is placed in an environment which recognizes the existence of a number of data areas on the backing store. In order to retain the freedom to rearrange the data areas for the purpose of reclaiming the areas no longer used, the areas are identified primarily by names chosen freely by the users. The current location of a named area is given in a catalogue of names and descriptions of areas which is a permanent part of the environment.

The areas of the backing store may be handled from Algol programs through calls of standard procedures of the system. The areas of the backing store are never rearranged while the execution of an Algol program is in progress. This makes it possible to allow the Algol program

to communicate with the areas of the backing store using absolute backing store track numbers for identification of the areas, thus saving a time consuming search of the catalogue at each transfer of data to or from the area. However, these absolute track numbers must be established anew in each execution of the Algol program. For this reason the communication with an area from an Algol program requires an initial reference to the catalogue, in addition to the actual transfers of data.

For communication with the catalogue during program execution, the user will call one of three standard procedures. A call of the form

reserve(⟨*name*⟩, ⟨*size*⟩)

will cause a reservation of an area with the given name and size to be made and entered in the catalogue. A call of the form

cancel(⟨*name*⟩)

will cancel the reservation of the area mentioned. The call

where(⟨*name*⟩, ⟨*area location*⟩)

will cause a search through the catalogue for the given name and an assignment to the area location of a description of the current place of storage of the area.

When the area location has been obtained through a call of *where*, transfers of blocks of data to or from the area are achieved by calls of two procedures, *put* and *get*. To place the values held in the array A in consecutive blocks of the area, starting at block 7 of that area the programmer must write

put(A, *area location*, 7) .

Calls of *get* have a similar form.

Programs in backing store.

The translator normally takes its input from paper tape. However, it is possible also to take programs, expressed as characters packed with six per word, partly or completely from the backing store. In this case the backing store may be also a magnetic tape. The area of the backing store must be identified by its name. To switch the translator from one medium to another, the program text must include symbols of the form

copy *name* <

which will cause the translator to continue taking its input from the area named. To switch back to the first medium, the second medium must contain the symbol **finis**.

Input and output procedures.

The built-in procedures for input and output were completely overhauled. The selection of character oriented media (paper tape, typewriter, and line printer) was made into a separate standard procedure, *select*. This achieves a match to the properties of the machine, which allows an independent simultaneous selection of several output media, each represented by one bit in a medium selection register.

The individual input and output functions were likewise changed to allow a good utilization of the machine. In fact, given the relatively low internal speed of the computer and the administration time necessary at each call of a procedure, much care is needed to avoid that the effective speed of input and output becomes out of proportion with the speed of the input and output devices. An additional factor is the automatic program segment administration, which puts an extra, indirect penalty on procedures which are so complicated that they require several tracks of instructions. These rather special circumstances, and the desire to make the system effective for general data processing, dictated the adoption of a set of input-output procedures based primarily on several fairly simple and very fast procedures. For a detailed description, see [1].

Machine language.

The new system allows instructions in machine language to be written directly among the Algol statements. Owing to the automatic handling of program segments during execution, two different treatments of the machine language instructions are offered. In the one, the machine language should be written as an Algol statement, and will be executed whenever the execution gets to that point of the program. In this treatment the machine language will take part in the automatic handling of segments. In the other treatment the machine language should be written like a declaration. Upon block entry this is transferred to the execution time stack, but not executed. The activation of it requires the call of a special standard procedure, *gier*. Stored in the stack this machine language program will always be present, irrespective of the automatic handling of segments.

Environment enquiries.

The system will be used on a number of different machine installations. To facilitate the writing of programs which adjust themselves to the particular properties of the machine actually at hand, a standard procedure, *system*, is included (cf. [6]). This will make available to the user a description of the actual machine configuration, including information about the storage capacity, special options, and peripheral units.

Gain in performance.

The performance gain resulting from these improvements depends greatly on the particular program. In favorable cases of programs which in the earlier versions of the system were squeezed tightly in the working store, and which in the new version can make use of the buffer store, a gain in speed of a factor of 5 has been reported. Users have also reacted very favorably to the new ease in handling parts of words.

Acknowledgements.

The design and development of the Gier Algol 4 system was a team effort. Besides the present author, the team members were: Tove Asmussen, Jørn Jensen, Søren Lauesen, Paul Lindgreen, Per Mondrup, and Jørgen Zachariassen.

REFERENCES

1. *A Manual of Gier Algol 4* (ed. P. Naur), A/S Regnecentralen, Copenhagen, 1967.
2. Naur, P., *The Design of the Gier Algol Compiler*, BIT 3 (1963), 124–140 and 145–166; also in Annual Review in Automatic Programming 4 (ed. R. Goodman), Pergamon Press 1964; for a translation into Russian, see Sovremennoye Programmirovanie, Izdatelstvo "Sovyetskoye Radio", Moskva 1966.
3. Naur, P., *The Performance of a System for Automatic Segmentation of Programs within an Algol Compiler (Gier Algol)*, Comm. ACM 8 (1965), 671–677.
4. Hoare, C. A. R., *Case expressions*, Algol Bulletin 18 (1964), 20–22.
5. Wirth, N. and Hoare, C. A. R., *A Contribution to the Development of Algol*, Comm. ACM 9 (June 1966), 413–432.
6. Naur, P., *Machine Dependent Programming in Common Languages*, BIT 7 (1967), 123–131.

A/S REGNECENTRALEN
COPENHAGEN
DENMARK